Trace Specifications of Non-Deterministic Multi-Object Modules

Michal Iglewski (e-mail: iglewski@uqah.uquebec.ca)
Département d'informatique, Université du Québec à Hull, Hull, Québec, Canada J8X 3X7

Marcin Kubica (e-mail: kubica@mimuw.edu.pl)
Jan Madey (e-mail: madey@mimuw.edu.pl)
Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland

ABSTRACT

The *Trace Assertion Method* (in short: TAM) is a formal method for abstract specification of interfaces of software modules being designed according to the "information hiding" principle. A trace specification is a "black-box" specification, i.e., it describes only those features of a module that are *externally observable*. The method was introduced by W. Bartusek and D.L. Parnas some 15 years ago and since then has undergone many modifications. In recent years there has been an increased interest in TAM. Software tools supporting practical usage of TAM for software engineering projects are under development, the method is being tested on different applications, its foundations are being studied.

Recent experiments with TAM have showed the need for further study in the case of non-deterministic multi-object modules. In this paper we investigate the expressiveness of the method for such modules. We present a formal model of a module and its TAM specification, show that the method requires some extensions and propose solutions. Our considerations are illustrated on TAM but could also be generally applied to modules with hidden non-determinism.

The full version of our investigations, including all definitions, lemmas, proofs and examples, is presented in university technical reports.

1 Introduction

The *Trace Assertion Method* (in short: TAM) is a formal method for abstract specification of interfaces of software *modules* being designed according to the "information hiding" principle [12]. A module implements one or more *objects*. A trace specification is a "black-box" specification, i.e., it describes only those features of an object that are *externally observable* and hides details of its internal structure. The method was first formulated in [1], and since then has undergone many modifications [3, 4, 7, 9, 11, 14, 15]. In recent years, there has been an increased interest in TAM, especially within the framework of the "functional approach" [13]. Software tools supporting practical usage of TAM are under development (e.g. [6, 15]), the method is being tested on different applications (e.g. [2, 9]), and its foundations are being studied (e.g. [8, 9, 15]).

Let us now justify our decision to study non-deterministic multi-object version of TAM. Firstly, a given module may be designed in TAM to implement either a single object or a number of objects. There is a distinct difference in the complexity of TAM

in these two cases. While introducing TAM we often limit our considerations to its single-object version, thus omitting certain problems (both syntactic and semantic) which arise in a more general case. In practical applications, however, a module is usually understood as an abstract data type, and a single-object module is treated as its special case, where there can be only one variable of that type. The present paper is a continuation and extension of [8] where TAM restricted to single-object modules was investigated and specifications were modelled as Mealy machines.

Secondly, non-determinism can be understood not only as a possible requirement (one may like to have a non-deterministic behavior of the program under development) but also as a certain philosophy in system design. A higher level of abstraction should leave as much freedom as possible to a specifier of a lower level which could be expressed in TAM by allowing non-deterministic modules. We think it is very important.

The structure of this paper is as follows. In Section 2 we present a formal model for multi-object modules. In Section 3 trace specifications for such modules are described. In Section 4 we show an example of a multi-object module that cannot be specified in the current version of TAM. A modified simple model of trace specifications which covers a wider class of non-determinism in multi-object modules is proposed in Section 5. Related changes in TAM and the expressiveness of a proposed version of TAM are discussed in Section 6. Final conclusions and future plans are briefly presented in Section 7.

2 A Formal Model for Multi-Object Modules

2.1 Introduction

We assume that time is discrete, linear, with an "initial" instant, and without a "final" one. Instants of time are represented by natural numbers.

The notion of an *object* can be characterized as follows. An object is any entity which has *states*, can be affected by *events*, and satisfies the following properties:

- at every instant of time the object is in one of its states; initially, the object is in the *initial* state,
- the object may change its state only as a result of an event; if the event occurs at the instant t, the object is in a new state at the instant $t + 1$.

Objects are grouped in modules: we say that a *module* implements a number of homogeneous and independent objects. If a module implements one object it is called a *single-object* module. A *multi-object* module can implement a given (including infinite) number of objects. In this paper we deal with multi-object modules.

Objects implemented by a given module are called *domestic*, while those implemented by other modules are called *foreign*. For each module there is a specific set of events that can affect its domestic objects.

There are two kinds of events that can affect an object:

- *access-program invocations* (calls of programs exported by the module),
- *input variable events* (changes of values of the module's input variables).

We assume that at most one event can occur at a given instant of time and that objects in the module can be affected only by a finite sequence of events.

In this paper we deal only with the first kind of events. However, we do not lose the generality of discussion since an input variable event can be expressed in terms of an access-program invocation. Input variables are described in detail in [2, 3, 14].

For each module there is a finite number of access-programs. Each access-program operates on at least one domestic object and possibly on foreign objects. As a result of an access-program invocation the arguments can change their states, possibly non-deterministically. For the sake of simplicity, we will treat values returned by functions as arguments which can change their states but with irrelevant initial values. For each access-program invocation, the next state of each argument depends only on the previous states of the arguments of this invocation. We assume that all arguments of access-programs are different objects. In practice, however, one object can be passed through several arguments of an access-program. This does not cause any loss of generality because we can model an access-program whose arguments might possibly represent the same objects by several access-programs whose arguments are always different objects. For example, an access-program P with two domestic arguments, which can represent the same object, can be modeled by two access programs:

- one with two domestic arguments (representing different objects passed as arguments of P), and
- one with only one domestic argument (representing one object passed through both arguments of P).

2.2 Modules

Def. 1 A *module* is the following tuple: (Q, q_0, O, F, I, E), where:

- Q is a non-empty set; its elements are called *states of domestic objects*,
- $q_0 \in Q$ and it is called the *initial state of domestic objects*,
- O is a non-empty set (possibly infinite); its elements are called *names of domestic objects*,
- F is a non-empty set; its elements are called *states of foreign objects*,
- I is a non-empty finite set; its elements are called *names of access-programs*,
- $E = (E_i)_{i \in I}$ is a sequence of relations $E_i \subseteq Q^{k_i} \times F^{l_i} \times Q^{k_i} \times F^{l_i}$ such that $k_i \in N$ (natural numbers), $1 \le k_i \le |O|$, and

$$\forall q \in Q^{k_i}, r \in F^{l_i} \exists q' \in Q^{k_i}, r' \in F^{l_i} [E_i(q, r, q', r')]$$

For each $i \in I$, E_i is a relation specifying changes of states of arguments of the access-program i. $E_i(q, r, q', r')$ means that if an invocation of the access-program i operates on certain domestic objects being in states q, and certain foreign objects being in states r, then the states of these objects after the invocation can be q', and r' respectively.

A tuple described in this definition can be treated as an extension of a Mealy machine — instead of a single state of an automaton we have many states of many ob-

jects.

Def. 2 A *state of a module* is a function $s: O \to Q$. It represents the states of all objects implemented by this module.

Def. 3 A *history of a module* is a function $H: N \to \wp(O \to Q) \setminus \{\emptyset\}$ (by $\wp(X)$ we denote the power set of X) such that $H(0) = \{s\}$, where $s(o) = q_0$ for all $o \in O$. It represents sets of possible states of this module at all instants of time. At the initial instant, all objects are in the initial state.

Def. 4 An *event* (i.e., an access-program invocation) is a tuple $e = (i, o, r)$, where:

- $i \in I$,
- $o \in O^{k_i}$ is a vector of different names of domestic objects; its elements identify domestic arguments of the event,
- $r \in F^{l_i}$ is a vector; its elements represent states of foreign arguments of the event.

Def. 5 An *output* of an event (i, o, r) is a vector $r' \in F^{l_i}$; its elements represent new states of foreign arguments of the event.

Def. 6 A *step of computation* is a pair $c = (e, r')$, where e is an event and r' is an output of e.

A step of computation represents externally (i.e. from outside of the module) observable aspects of an access-program invocation:

- which access-program is invoked,
- on which domestic objects the access-program operates,
- what are the states of passed foreign arguments, and
- what are the states of foreign arguments after the invocation.

Def. 7 A state s' of a module is *reachable* from a state s of the module in a step of computation $c = ((i, o, r), r')$ iff:

$$E_i\left(\left(s(o_1), \ldots, s\left(o_{k_i}\right)\right), r, \left(s'(o_1), \ldots, s'\left(o_{k_i}\right)\right), r'\right) \wedge$$
$$\forall j \in O \setminus \{o_1, \ldots, o_{k_i}\}\ [s(j) = s'(j)]$$

We denote it by $s \xrightarrow{c} s'$.

For a given $c = (e, r')$, $s \xrightarrow{c} s'$ means that if a module is in a state s, an event e takes place and new states of its foreign arguments are equal to r', then a new state of the module can be s'.

Def. 8 A *computation* is a finite sequence of steps of computation, $((e_j, r'_j))_{j=0}^{m}$, where $m \geq -1$, $e_j = (i_j, o_j, r_j)$ is an event, and $r'_j \in F^{l_{i_j}}$ is an output of e_j.

A computation denotes externally observable aspects of a sequence of invocations of access-programs. For $m = -1$ this definition denotes an empty sequence of invocations.

Def. 9 We say that a history H *satisfies a computation* $C = (c_j)_{j=0}^{m}$ iff for all $t \in N$:

$$\left(t \le m \;\Rightarrow\; H(t+1) = \{s' : O \to Q \mid \exists s \in H(t) \left[\, s \xrightarrow{c_t} s' \,\right]\}\right) \wedge$$
$$(t > m \Rightarrow H(t+1) = H(t)) .$$

A history represents possible states of domestic objects during the computation. Notice that if there exists a history satisfying a given computation then there is only one such history.

Def. 10 We say that a computation is *feasible* if there exists a history satisfying this computation.

One should recall that the value of a history at a given instant of time is a non-empty set of possible states of the module. Hence, there can be (and usually are) computations that are not feasible.

The set of feasible computations fully characterizes an externally observable behavior of a module. According to the information-hiding principle, we often deal only with externally observable aspects of a module, i.e., we are not interested in the concrete states of domestic objects — we observe only the identity of domestic arguments and the values of foreign arguments of events. Hence, there can be several modules that cannot be externally distinguished.

Def. 11 We say that two modules are *observationally equivalent* iff they have the same sets of feasible computations.

Sometimes we are interested in reducing a module to a simpler, observationally equivalent module. A simple reduction can be done by removing states which can never appear.

Def. 12 The set A of *reachable states of objects* of a module $M = (Q, q_0, O, F, I, E)$ is the subset of Q of all states appearing in histories satisfying feasible computations:

$$A = \{q \in Q \mid \exists C\text{-feasible}, H\text{-satisfying } C, t \in N, s \in H(t), o \in O\ [q = s(o)]\}$$

Notice that always $q_0 \in A$. Non-reachable states of a module do not appear in any history satisfying any feasible computation. Hence, they are irrelevant to the behavior of the module.

3 Trace Specifications of Multi-Object Modules

3.1 Trace-Modules

Def. 13 A *trace-module* is a module (Q, q_0, O, F, I, E) such that for each $i \in I$ there exist:

- a relation $R_i \subseteq Q^{k_i} \times F^{l_i} \times F^{l_i}$,
- a function $X_i : R_i \to Q^{k_i}$,

such that:
$\forall q, q' \in Q^{k_i}, r, r' \in F^{l_i}\ [E_i(q, r, q', r') \Leftrightarrow R_i(q, r, r') \wedge X_i(q, r, r') = q']$.
R_i is called a *return relation* (between values of arguments of an access-pro-

gram invocation and its output); X_i is called an *extension function* and describes the values of domestic arguments after this invocation, depending on the values of arguments and the output of the invocation.

Intuitively, a module (Q, q_0, O, F, I, E) is a trace-module iff for each $i \in I$ the relation E_i can be viewed as a composition of a relation (R_i) denoting new states of foreign arguments and a function (X_i) denoting new states of domestic arguments. Generally, states of trace-modules have simpler forms than states of modules — for each feasible computation and an instant of time, there is only one possible state of a trace-module.

Lemma 1: If M is a trace-module, C is a computation, H is a history satisfying C, and $t \in N$, then $H(t)$ is a singleton.

A proof of this lemma can be found in [5].

One should note that if the given module is deterministic, then it is also a trace-module, since for each $i \in I$, E_i is a function.

3.2 Traces

One of the basic notions in TAM is the notion of traces. Intuitively, a *trace* is a term describing a fragment of a computation of a trace-module, containing all steps of that computation that can influence the current state of a given object.

Def. 14 The set of *syntactically correct* traces of a trace-module $M = (Q, q_0, O, F, I, E)$ is the smallest set such that:

1. the empty trace, denoted by "_", is a syntactically correct trace, and
2. if $i \in I$, $r, r' \in F^{l_i}$, $T_1, \ldots, T_{k_i}$ are syntactically correct traces, then for each $1 \le j \le k_i$ the following term:

$$T_j.i\left(T_1, \ldots, T_{j-1}, *, T_{j+1}, \ldots, T_{k_i}, r, r'\right) \text{ if } T_j \neq \_ \text{, or}$$

$$i\left(T_1, \ldots, T_{j-1}, *, T_{j+1}, \ldots, T_{k_i}, r, r'\right) \text{ if } T_j = \_$$

is a syntactically correct trace.

This definition of syntactically correct traces is simplified, according to our model of modules (cf. Section 2.1). Detailed descriptions of traces can be found in [7, 14].

We use a dot (".") as an operator of concatenation of syntactically correct traces with the empty trace being neutral element (for each syntactically correct trace T, $T.\_ = \_.T = T$).

Not all syntactically correct traces denote fragments of feasible computations of a trace-module.

Def. 15 Let $C = (c_i)_{i=0}^{m}$ be a feasible computation of a trace-module $M = (Q, q_0, O, F, I, E)$, H be a history satisfying C, $t \in N$ be an instant of time and $o \in O$ be an object. *A syntactically correct trace T representing the state of object o at instant t in computation C* (from lemma 1 we know that $H(t)$ is a singleton, and hence, there is only one such state) is defined as follows:

1. if $t = 0$ then $T = \_$, and
2. if $0 < t \leq m + 1$ then let $c_{t-1} = \left(\left(i, \left(o_1, \ldots, o_{k_i}\right), r\right), r'\right)$:
   - if $o \notin \{o_1, \ldots, o_{k_i}\}$, then T is equal to the trace representing the state of object o at instant $t - 1$,
   - if $o = o_j$, $1 \leq j \leq k_i$, $T_1, \ldots, T_{k_i}$ are traces representing the states of objects $o_1, \ldots, o_{k_i}$ at instant $t - 1$, then:

     $T = T_j.i\left(T_1, \ldots, T_{j-1}, *, T_{j+1}, \ldots, T_{k_i}, r, r'\right)$, and
3. if $t > m + 1$, then T is equal to the trace representing the state of object o at instant $m + 1$.

We also say that a syntactically correct trace T *represents a state* q if a feasible computation C, a history H satisfying C, an object $o \in O$ and an instant of time $t \in N$ exist such that $H(t) = \{s\}$ and $q = s(o)$.

Note that there can be many traces representing one state, but one syntactically correct trace can represent at most one state. One should also note that a state is represented by one or more traces iff it is a reachable state.

Def. 16 A syntactically correct trace of a trace-module is *feasible* iff it represents a state.

For each module, the empty trace ("_") is always feasible and represents the initial state of every object.

Further on in this section by "traces" we mean always "feasible traces". In TAM, states of domestic objects are represented by traces. However, a state is often represented by many traces.

Def. 17 Two traces T_1 and T_2 are *equivalent* ($T_1 \equiv T_2$) iff they represent the same state.

Notice that "≡" is an equivalence relation. Thus we can represent states of domestic objects by the equivalence classes of "≡". In TAM we do not use the equivalence classes as such but we represent states of objects by the fixed representatives of the equivalence classes. These representatives are called *canonical traces*.

3.3 Trace Specifications

The goal of this paper is to study specifications of non-deterministic, multi-object modules. Without sacrificing generality, we will limit our considerations to non-parameterized specifications only. (Since the semantics of a parameterized specification for the given actual parameters is a non-parameterized specification, all our observations apply to parameterized specifications also).

A trace specification (i.e., a specification in TAM) of a module is a document consisting of the following five parts: Characteristics Section, Syntax Section, Canonical Section, Equivalence Section, Return Values Section. Precise descriptions of trace specifications can be found e.g. in [7, 14]. Here, we only briefly summarize the contents of those sections. Example specifications can be found in [7, 9].

The Characteristics Section contains information about:

- the name of the module specified by the given specification,
- foreign modules used by this module; they implicitly define the set of states of foreign objects,
- the set of names of objects implemented by the module, and
- features of the module (e.g, whether it is single-object or multi-object).

The Syntax Section defines the set of access-programs and the types of their arguments. In particular, for each access-program, the Syntax Section defines the number of domestic and foreign arguments. The Syntax Section provides some information that is not expressed in our model, e.g., the order of domestic and foreign arguments for each access-program. In our model this order is fixed. Nor does our model distinguish types of foreign arguments. One should note that the Syntax Section implicitly defines the set of names of access-programs and the set of syntactically correct traces.

The Canonical Section defines the characteristic predicate (*canonical)* of the set of canonical traces. If the empty trace is not canonical, then this section explicitly defines a canonical trace representing the initial state. The set of canonical traces depends on the particular specification. In the rest of the specification, states of domestic objects are represented by canonical traces, i.e., the set of states of domestic objects is the set of canonical traces. This section of the specification can also define some auxiliary functions and/or relations used in the rest of the specification.

For each access-program, the Equivalence Section contains a definition of the *extension function*. The domain of this function contains states of all arguments of the access-program before the invocation and new states of all of its foreign arguments after the invocation. The range of this function contains new states of all domestic arguments of the access-program, and a sort of a marker (called *a token*) describing the correctness of the invocation. This marker is not expressed in our model. The correctness of the invocation has no effect on the behavior of the module. In the rest of this paper we will skip this aspect of extension functions. One can assume that every invocation is specified as a correct one.

The Return Values Section for each access-program defines a relation called the *return relation*. This is a relation between states of all arguments of the access-program before the invocation, and new states of all of its foreign arguments. This relation determines possible new states of foreign arguments of the access-program.

A trace specification can be modelled by a trace-module as follows:

- Q is the set of canonical traces,
- $q_0 \in Q$ is the canonical trace representing the initial state,
- O is the set of names of objects implemented by the module,
- F is the union of sets of canonical traces of foreign modules,
- I is the set of names of access-programs,
- E_i is such that:

$$\forall q, q' \in Q^{k_i}, r, r' \in F^{l_i} [E_i(q, r, q', r') \Leftrightarrow R_i(q, r, r') \wedge X_i(q, r, r') = q']$$

where R_i is a return relation and X_i is an extension function for an access-program i.

A trace-module thus defined is called the *trace-module obtained from a trace specification*.

Def. 18 We say that a *module satisfies a trace specification* iff it is observationally equivalent to the trace-module obtained from the trace specification. In this case we also say that the *specification specifies the module*.

We are not only able to represent trace specifications by trace-modules but we can specify every trace-module, which is more interesting.

Theorem: For each trace-module there exists a trace specification satisfied by this module.

A proof of this theorem can be found in [5].

This theorem proves that the class of modules that can be specified in TAM is equal to the class of modules observationally equivalent to certain trace-modules. One should also note that since every deterministic module is a trace-module, every deterministic module can be specified in TAM.

4 Non-Determinism Non-Expressible in the Trace Assertion Method

In this section we prove (by providing a counter-example) that not every non-deterministic multi-object module can be specified in TAM.

Theorem: There exists a module that does not satisfy any trace specification.

Proof: Let $M = (Q, q_0, O, F, I, E)$ be a module, where:

- $Q = \wp(\{0, 1\})$,
- $q_0 = \emptyset$,
- $O = \{a, b\}$,
- $F = \{0, 1, true, false\}$,
- $I = \{Ins, In, Cross\}$,
- $E_{Ins} \subseteq Q \times F \times Q \times F$,
  $E_{Ins}(q, r, q', r') \equiv r' = r \wedge q' = q \cup (\{r\} \cap \{0, 1\})$,
- $E_{In} \subseteq Q \times F \times Q \times F$, $E_{In}(q, r, q', r') \equiv q = q' \wedge r' = (r \in q)$,
- $E_{Cross} \subseteq Q^2 \times Q^2$,
  $E_{Cross}(q_1, q_2, q'_1, q'_2) \equiv q_1 \cup q_2 = q'_1 \cup q'_2 \wedge q'_1 \cap q'_2 = \emptyset$.

The module M implements two sets that both can contain two elements, 0 and 1, with the following operations:

- *Ins* inserts an element into a set,
- *In* checks if an element is in a set,
- *Cross* takes two sets and divides non-deterministically their union into two disjoint sets.

We will show that there does not exist a trace specification satisfied by M.

The proof is by contradiction. Let us assume that such a trace specification X exists. Let $\overline{M} = \left(\overline{Q}, \bar{q}_0, O, F, I, \overline{E} \right)$ be a trace-module obtained from X.

Let us consider the following computations:

$C_1 = (((Ins, a, 1), 1), ((Cross, a, b)), ((In, a, 1), true), ((In, b, 1), false)),$

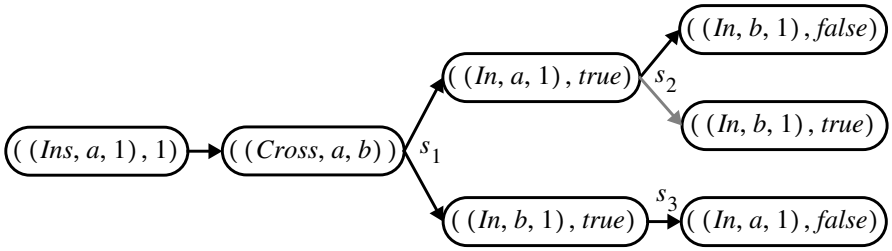
$C_2 = (((Ins, a, 1), 1), ((Cross, a, b)), ((In, b, 1), true), ((In, a, 1), false)),$

$C_3 = (((Ins, a, 1), 1), ((Cross, a, b)), ((In, a, 1), true), ((In, b, 1), true)).$

C_1 and C_2 are feasible computations for M and hence also for $\overline{M}$ but C_3 is not feasible for M. This means that when we insert 1 into set a, and then apply the access-program *Cross* to sets a and b, 1 is in one of these sets but not in both.

We will obtain the contradiction by proving that the computation C_3 is feasible for module $\overline{M}$.

Let H_1, H_2 be two histories of a trace-module $\overline{M}$ satisfying, respectively, computations C_1 and C_2. H_1 and H_2 have the same first two elements and hence, $H_1(2) = H_2(2)$. From lemma 1, $H_1(t)$ and $H_2(t)$ are singletons (for each $t \in N$). Let $H_1(2) = H_2(2) = \{s_1\}$, $H_1(3) = \{s_2\}$, and $H_2(3) = \{s_3\}$.

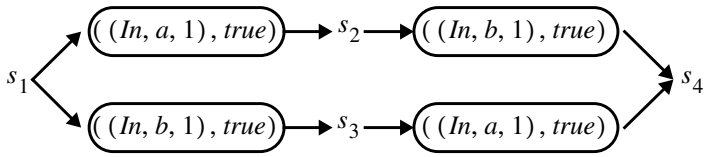


Notice that $s_1(b) = s_2(b)$ because $C_{1,2} = C_{3,2} = ((In, a, 1), true)$ cannot change the state of object b, and $s_1(a) = s_3(a)$ because $C_{2,2} = ((In, b, 1), true)$ cannot change the state of object a.

To prove that C_3 is feasible for $\overline{M}$ we show that for $C_{3,3} = ((In, b, 1), true)$ there exists a state s_4 of a trace-module $\overline{M}$ such that $s_2 \xrightarrow{C_{3,3}} s_4$. Notice that $C_{3,3}$ cannot change the value of object a, so $s_4(a) = s_2(a)$. On the other hand, $s_4(b) = s_3(b)$ because $s_2(b) = s_1(b)$ and $C_{3,3} = C_{2,2} = ((In, b, 1), true)$. Hence, state s_4 can be defined as follows:

$$s_4(j) = \begin{cases} s_2(a) & \text{if } j = a \\ s_3(b) & \text{if } j = b \end{cases}$$

Notice that for $z = ((In, a, 1), true)$ also $s_3 \xrightarrow{z} s_4$.



A history H_3 satisfying computation C_3 is defined as follows:

$$H_3(t) = \begin{cases} H_1(t) & \text{if } t \le 3 \\ \{s_4\} & \text{if } t > 3 \end{cases}$$

Thus, computation C_3 is feasible for $\overline{M}$ but not feasible for M. Module $\overline{M}$ is not observationally equivalent to M, and M does not satisfy X. ■

5 "New" Trace-Modules

We will now redefine the notion of trace-modules in such a way, that for every module there exists an observationally equivalent "new trace-module". This reduces the problem of specification of multi-object modules in TAM to that of specification of new trace-modules.

Def. 19 A *new trace-module* is such a module (Q, q_0, O, F, I, E) that for each E_i there exist:

- a number $1 \le p_i \le k_i$,
- a relation $R_i \subseteq Q^{k_i} \times F^{l_i} \times Q^{k_i - p_i} \times F^{l_i}$,
- a function $X_i : R_i \to Q^{p_i}$,

such that $\forall q \in Q^{k_i}, r \in F^{l_i} \exists q' \in Q^{k_i - p_i}, r' \in F^{l_i} [R_i(q, r, q', r')]$, and

$$\forall q, q' \in Q^{k_i}, r, r' \in Q^{l_i} \left[E_i(q, r, q', r') \Leftrightarrow R_i\left(q, r, \left(q'_{p_i+1}, \ldots, q'_{k_i}\right), r'\right) \wedge X_i\left(q, r, \left(q'_{p_i+1}, \ldots, q'_{k_i}\right), r'\right) = \left(q'_1, \ldots, q'_{p_i}\right) \right]$$

R_i is called a *return relation*, X_i is called an *extension function* and p_i is called a *number of primary domestic arguments* of access-program i. The largest p_i is called the *maximum number of primary domestic arguments* of access-program i.

Intuitively, in new trace-modules we allow new states of some domestic arguments to be specified by the return relation. One should note that for a given new trace-module and the numbers of primary domestic arguments, for each access-program there exist exactly one return relation and one extension function.

Each trace-module is also a new trace-module (for $p_i = k_i$); however, the class of new trace-modules is wider than the class of trace-modules.

Theorem: For each module M, there exists a new trace-module $\overline{M}$ observationally equivalent to M.

A proof of this theorem can be found in [5].

6 "New" Trace Specifications

The basic difference between new trace-modules and trace-modules is in the definition of the extension function:

- in trace-modules, an extension function determines new states of all domestic arguments of an access-program,
- in new trace-modules, an extension function defines only new values of some (at least one) domestic arguments, called *the primary (domestic) arguments*; possible new values of the rest of domestic arguments, called *the secondary (domestic) arguments*, are defined by the return relation.

This difference is reflected by including within a new-trace, new-traces that denote new states of the secondary domestic arguments. It also changes the interpretation of traces. A trace represents a new value of one of the primary domestic arguments of its last invocation.

Def. 20 Let $M = (Q, q_0, O, F, I, E)$ be a new trace-module and $(p_i)_{i \in I}$ be the maximum numbers of primary domestic arguments of access-programs. The set of *syntactically correct new-traces of M* is the smallest set such that:

1. the empty trace, denoted by "_", is a syntactically correct new-trace, and
2. if $i \in I$, $1 \le j \le u \le p_i$, $r, r' \in F^{l_i}$, $T_1, \ldots, T_{k_i}$ and $U_{u+1}, \ldots, U_{k_i}$ are syntactically correct new-traces, then the following term:

$$T_j \cdot i\left(T_1, \ldots, T_{j-1}, *, T_{j+1}, \ldots, T_{k_i}, r, U_{u+1}, \ldots, U_{k_i}, r'\right) \text{ if } T_j \neq \_, \text{ or}$$

$$i\left(T_1, \ldots, T_{j-1}, *, T_{j+1}, \ldots, T_{k_i}, r, U_{u+1}, \ldots, U_{k_i}, r'\right) \text{ if } T_j = \_$$

is a syntactically correct new-trace.

Numbers of primary arguments influence the form of new-traces in such a way that a new-trace includes sub-new-traces describing new states of all secondary arguments of access-program invocations. This definition allows all possible numbers of primary arguments, i.e., from 1 to the maximum number of primary arguments ($1 \le u \le p_i$). However, the simplest, not redundant form is for the maximum number of primary arguments ($u = p_i$).

The definition of feasibility of new-traces and the definition of feasible new-traces representing states are more complex and can be found in [5].

We allow new-traces with different numbers of primary arguments, however the most expressible are new-traces with the maximum number of primary arguments. They contain the fewest sub-new-traces representing new states of secondary domestic arguments, and they can represent new states of the largest number of domestic arguments. Each new-trace with the number of primary arguments less than the maximum can be easily transformed to one with the maximum number of primary arguments by removing some of sub-new-traces denoting new states of secondary domestic arguments.

The equivalence relation on feasible new-traces ("≡") and the set of canonical new-traces can be defined the same way as for traces. However, it can happen that some of reachable states cannot be represented by any feasible new-trace. We discuss

the consequences of this fact later on.

Def. 21 We say that a state $q \in Q$ of an object implemented by a new trace-module $M = (Q, q_0, O, F, I, E)$ is *trace-expressible* if there exists a feasible new-trace representing q.

The form of a new trace specification is similar to the form of a trace specification. The form of the Characteristics Section is the same. The Syntax Section defines also numbers of primary arguments of access-programs. It defines implicitly the subset of new-traces used in the specification. The Canonical Section defines a set of canonical new-traces and a canonical new-trace representing the initial state. The Equivalence Section defines an extension function for each access program. The range of this function contains only canonical new-traces, representing new states of the primary arguments. The Return Values Section defines a return relation for each access program. However, this relation is defined on states of all arguments passed to the access-program, and new states of all secondary domestic arguments and all foreign arguments of the access-program.

A new trace specification can be modelled as a new trace-module (Q, q_0, O, F, I, E) in the following way:

- Q is the set of canonical new-traces,
- q_0 is the canonical new-trace representing the initial state,
- O is the set of names of objects implemented by the module,
- F is the union of sets of canonical traces of all foreign modules,
- I is the set of names of access programs,
- for each $i \in I$, E_i is a relation defined as follows:

$$\forall q, q' \in Q^{k_i}, r, r' \in F^{l_i} \left[E_i(q, r, q', r') \Leftrightarrow R_i\left(q, r, r', \left(q'_{p_i+1}, \ldots, q'_{k_i}\right)\right) \wedge X_i\left(q, r, r', \left(q'_{p_i+1}, \ldots, q'_{k_i}\right)\right) = \left(q'_1, \ldots, q'_{p_i}\right) \right]$$

An example specification of the module with the "Cross" access-program (cf. Section 4) can be found in [5].

It turns out that not all new trace-modules can be specified in the proposed version of TAM. It can happen that some of the reachable states are not trace-expressible. In such a case we are simply unable to represent these states in the specification. But if all reachable states of a new trace-module are trace-expressible then we can specify such a module in the proposed version of TAM.

Theorem: Let $M = (Q, q_0, O, F, I, E)$ be a new trace-module. If all reachable states of M are trace-expressible, then there exists a new trace specification satisfied by M.

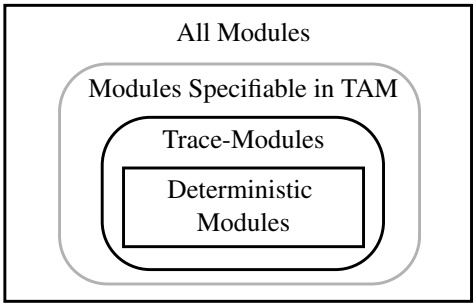
A proof of this theorem can be found in [5].

A class of modules which can be specified in the proposed version of TAM is the class of modules observationally equivalent to some new trace-modules having all (reachable) states trace-expressible.

7 Conclusions

The main goal of this paper was to investigate the expressiveness of TAM. As it was proved in [8], every single-object module can be specified in TAM. Also every deterministic multi-object module can be specified in TAM. However, there exists a non-deterministic multi-object module which cannot be specified in TAM.

We have defined a sub-class of modules (called trace-modules) which effectively characterizes the expressiveness of TAM. Each trace-module can be specified in TAM and each specification can be modelled by a trace-module. Hence, the class of multi-object modules which can be specified in TAM is the class of modules observationally equivalent to some trace-modules. This situation can be illustrated by the following diagram:



We have also proposed some modifications in TAM and we have defined an appropriate sub-class of modules (called new trace-modules) to model specifications in the proposed version of TAM. It is a property of this class that each module is observationally equivalent to some new trace-module. We have extended the expressiveness of TAM, although we have not covered the whole class of non-deterministic multi-object modules. Not all new trace-modules can be specified in the proposed version of TAM because it can happen that some (reachable) states of objects implemented by the module cannot be expressed by traces.

If we could express by traces all (reachable) states of objects implemented by new trace-modules then we would be able to specify all modules in TAM. This problem still limits the usefulness of TAM in specification of some non-deterministic multi-object modules. Extension of the expressiveness of traces, to cover all reachable states of new trace-modules, will be one of our goals in future research.

Acknowledgements

This paper emerged from many discussions on TAM we have had in Hamilton (with D.L. Parnas and his group), in Hull, and in Warsaw. We are very grateful to all our colleagues for their contribution, and especially to J. Mincer-Daszkiewicz and K. Stencel.

This work was partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), by the State Committee for Scientific Research in Poland (KBN, grant 8 S503 040 04), by Digital Equipment's European External Research Programme (EERP PL-002), and by NATO Linkage grant (HTECH.LG.941314).

References

1. Bartussek, W., Parnas, D.L., "Using Traces to Write Abstract Specifications for Software Modules", in Gehani, N., McGettrick, A.D. (Eds.), *Software Specification Techniques,* AT&T Bell Telephone Laboratories, 1985, pp. 111-130.
2. Bojanowski, J., Iglewski, M., Madey, J., Obaid, A., "Functional Approach to Protocol Specification", in *Protocol Specification, Testing and Verification XIV*, Vuong, S.T., Chanson, S.T. (Eds.), Chapman & Hall, 1995, pp. 395-402.
3. Erskine, N., "The Usefulness of the Trace Assertion Method for Specifying Device Module Interfaces", *CRL Report* No. 258, McMaster Univ., CRL, Telecommun. Res. Inst. of Ontario (TRIO), Hamilton, Ont., Canada, 1992.
4. Hoffman, D.M., "The Trace Specification of Communications Protocols", *IEEE Trans. on Computers*, Vol. C-34, No. 12, Dec.r 1985, pp. 1102-1113.
5. Iglewski, M., Kubica, M, Madey, J., "Trace Specifications of Non-deterministic Multi-object Modules", *Technical Report* TR 95-05 (205), Warsaw Univ., Inst. of Informatics, Warsaw, Poland, 1995.
6. Iglewski, M., Kubica, M, Madey, J.,"Editor for the Trace Assertion Method", in: *Proc. 10th Int. Conf. of CAD/CAM, Robotics and Factories of the Future: CARs & FOF'94,* Zaremba, M. (Ed.)*,* OCRI, Ottawa, Ont., Canada, 1994, pp. 876-881.
7. Iglewski, M., Madey, J., Parnas, D.L., Kelly, P.C., "Documentation Paradigms", *CRL Report* No. 270, McMaster Univ., CRL, Telecommun. Res. Inst. of Ontario (TRIO), Hamilton, Ont., Canada, 1993.
8. Iglewski, M., Madey, J., Stencel, K., "On Fundamentals of the Trace Assertion Method", *Technical Report* TR 94-09 (198), Warsaw Univ., Inst. of Informatics, Warsaw, Poland, 1994.
9. Iglewski, M., Mincer-Daszkiewicz, J., Stencel, K., "Some Experiences with Specification of Non-deterministic Modules", *Technical Report* RR 94/09-7, Université du Québec à Hull, Département d'Informatique, Hull, Québec, Canada, 1994.
10. Iglewski, M., Kubica, M., Madey, J., Mincer-Daszkiewicz, J., Stencel, K., "Report of the Trace Assertion Method 95", *in preparation.*
11. McLean, J.D., "A Formal Foundation for the Abstract Specification of Software", *Journal of the ACM*, Vol. 31, No. 3, July 1984, pp. 600-627.
12. Parnas, D.L., "On the Criteria to be used in Decomposing Systems into Modules", *Commun. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.
13. Parnas, D.L., Madey, J., "Functional Documents for Computer Systems", *Science of Computer Programming*, to appear in 1995.
14. Parnas, D.L., Wang, Y., "The Trace Assertion Method of Module Interface Specification", *Technical Report* 89-261, Queen's Univ., C&IS, Telecommun. Res. Inst. of Ontario (TRIO), Kingston, Ont., Canada, 1989.
15. Wang, Y., "Specifying and Simulating the Externally Observable Behavior of Modules", (Ph.D. Thesis)*, CRL Report* No. 292, McMaster Univ., CRL, Telecommun. Res. Inst. of Ontario (TRIO), Hamilton, Ont., Canada, 1994.